IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re application of: Christopher Wilson et al.	Art Unit: 2141
Appl. No. 10/748,723	Examiner: Quang N. Nguyen
Filed: December 30, 2003	Atty. Docket: 07942.0005.CPUS03
For: METHOD, SYSTEM AND APPARATUS FOR MESSAGING BETWEEN WIRELESS MOBILE TERMINALS AND NETWORKED COMPUTERS	

DECLARATION OF CHRISTOPHER R. D. WILSON UNDER 37 C.F.R § 1.131

Mail Stop RCE
Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

I, Christopher R. D. Wilson, hereby declare:

1. I am over the age of eighteen years, and, except for matters identified as being based on information and belief, have personal knowledge of the matters stated herein. If called upon to do so, I would testify as a witness to these matters.

2. All statements made herein on the basis of personal knowledge are true, and all statements made herein on the basis of information and belief are believed to be true.

3. I have a Bachelor degree in Electrical Engineering.

4. I am a System Architect for Fastmobile, Inc. ("Fastmobile"), the assignee of the above-referenced patent application, and I have been employed by Fastmobile since February 1, 2002.

5. I am a joint inventor named in the above-referenced patent application. I have reviewed and understand the contents of the patent application and the Office Action dated February 8, 2007. I have particularly reviewed claims 24 – 28 and 45 – 55 as they are currently amended.

6. I understand that claims 24 – 28 and 45 - 55 are rejected under 35 U.S.C. 102(e) as being anticipated by Koskelainen et al. (2004/0224710) ("Koskelainen").

7. Fastmobile had already built, operated and tested a wireless chat system with all of the features recited in claims 24 – 28 and 45 - 55 prior to May 7, 2003, the filing date of the Koskelainen patent application.

8. The wireless chat system was built at Fastmobile's facilities in Schaumburg, Illinois.

9. Prior to May 7, 2003, I was part of the engineering team that developed Fastmobile's aforementioned wireless chat system. I personally developed and implemented the server software for the system with other engineers.

10. Prior to May 7, 2003, the wireless chat system included a Dell Power Edge 2450 server running Red Hat Linux Version 7.2 and Java version 1.2. The Dell server executed chat server software. Some of the source code for the chat server software is attached as Exhibit 1. The wireless chat system also included cellular phones, e.g., Nokia 3650 running Symbian OS 6.1 and executing wireless chat client software; and personal computers (PCs) executing PC chat client software. The PCs were running

the Microsoft Windows 2000 operating system. Some of the source code for the wireless chat client software is attached as Exhibit 2; and some of the source code for the PC chat client software is attached as Exhibit 3.

11. The cellular phones were on the T-Mobile network. The PCs were connected to a wired Ethernet LAN in Fastmobile's offices, which was connected to the Internet. The Power Edge server was also connected to Fastmobile's Ethernet LAN and the Internet. The server communicated with the carrier network through the Internet. The server and cellular phones communicated through the carrier network via the Internet. The server and PCs communicated over the Internet.

12. The chat server software was developed, coded and tested prior to May 7, 2003 (see Exhibit 1 at page 1). Exhibit 1 contains a portion of the actual chat server software code. The server software code was written in Java. When executed, the server software code configured the server to permit chatting between cellular phones and networked PCs. The chat server software code establishes chat sessions in response to login requests from wireless and PC clients. The server software code also causes the server to forward client messages to recipient clients and store client messages when the intended recipient is temporarily unavailable to receive messages through the system. The server software causes the server to forward certain messages to separate email and IM servers, and also causes the server to store a list of message recipients. In addition, the chat server software processes text and voice messages, including streaming voice.

13. The wireless chat client software was developed, coded and tested prior to May 7, 2003 (see Exhibit 2 at page 1). Exhibit 2 contains a portion of the actual wireless chat client software code. The wireless chat client software code was written in Java and

was executed by certain cellular phones running the Symbian operating system. When executed, the wireless chat client software permits a cellular phone to chat with other wireless devices and networked PCs, by way of the chat server. When executed, the wireless chat client software can send login requests to the chat server, retrieve buddy lists (recipient lists) from the server, present graphic user interfaces on the phone for composing text messages and selects message recipients, and provides push-to-talk functionality at the phone for sending messages. In addition, the wireless chat client software processes text and voice messages, including streaming voice, and presents graphic user interfaces for recording voice messages, listening to received voice messages, and displaying text messages.

14. The PC chat client software was developed, coded and tested prior to May 7, 2003 (see Exhibit 3 at page 1). Exhibit 3 contains a portion of the actual PC chat client software code. The PC chat client software code was written in C++ and was executed by certain PCs running the Microsoft Windows 2000 operating system. When executed, the PC chat client software permits a networked PC to chat with other PCs and wireless devices, e.g., cellular phones, by way of the chat server. When executed, the PC chat client software can send login requests to the chat server, retrieve buddy lists (recipient lists) from the server, present graphic user interfaces on the PC for composing text messages and selects message recipients, and provide push-to-talk functionality at the PC for sending messages. In addition, the PC chat client software processes text and voice messages, including streaming voice, and presents graphic user interfaces for recording voice messages, listening to received voice messages, and displaying text messages.

15. I am aware that willful false statements and the like are punishable by fine or imprisonment, or both (18 U.S.C. § 1001).

I declare under penalty of perjury that the foregoing is true and correct. Executed on this 6 day of July, 2007 at Rolling Meadows, Illinois.

CHRISTOPHER R. D. WILSON

```
*****************  Version 20  *****************
User: Nick Southwell  Date:  7/23/02  Time:  2:13p
Checked in $/Version1/server/src/com/MTalk/YoChat/Transport
Comment:
  Replace code needed by disabled unit test

*****************  Version 19  *****************
User: Chris Wilson    Date:  7/19/02  Time:  5:10p
Checked in $/Version1/server/src/com/MTalk/YoChat/Transport
Comment:
  Added Reply All to BeginAudio..
- Made IPaq Hack configurable from Properties file

*****************  Version 18  *****************
User: Chris Wilson    Date:  7/18/02  Time:  2:10p
Checked in $/Version1/server/src/com/MTalk/YoChat/Transport
Comment:
  Still TODO: Added Unit Tests for Audio and Text Messages
- Changed TextMessages to work for reply all.
- Also reworked Messages to be more efficient with HA

*****************  Version 17  *****************
User: Chris Griffin   Date:  7/11/02  Time:  2:05p
Checked in $/Version1/server/src/com/MTalk/YoChat/Transport
Comment:
  update for smp
-----------------------------------------------------------------------------

//BeginClientToServerAudioMessage.java------------------------------------------
--
package com.MTalk.YoChat.Transport;

import java.util.*;
import java.net.*;
import java.io.*;
import com.MTalk.YoChat.ejb.*;
import com.MTalk.YoChat.util.*;
import com.MTalk.YoChat.util.ServiceLocatorException;
import java.rmi.RemoteException;
import javax.ejb.FinderException;

public class BeginClientToServerAudioMessage extends ServerMessage
{
    private static final int TYPE =
        YcProtocolMessageType.BEGINCLIENTTOSERVERAUDIO;

    public boolean m_groupsOrBuddies = false;
    public int[] m_ids = null;
    private int m_authorId = 0;
    private int m_voiceSessionId = -1;

    public static void log(int errLevel, String msg)
    {
        Debug.LogIt(errLevel, ILogger.AUDIO, "[Begin] " + msg);
    }

       public static ServerMessage create(String sAddress, int port,
                                        int sequenceNumber, int sessionId,
                                        boolean groupsOrBuddies, int[] ids)
                                        throws UnknownHostException
       {
        try
        {
            ServerMessage srvrMsg = new ServerMessage(sAddress, port,
                    sequenceNumber, sessionId,
                    YcProtocolMessageType.BEGINCLIENTTOSERVERAUDIO);
            DataOutputStream out = srvrMsg.getDataOutput();
```

Exhibit
1

```
            out.writeByte((byte)(groupsOrBuddies?1:0));                        //
group or buddy flag
            if (ids != null)
            {
                out.writeByte((byte)ids.length);
                for (int i = 0; i < ids.length; ++i)
                {
                    out.writeInt(ids[i]);
                }
            }
            else
            {
                out.writeByte(0);       //Number of groupsOrBuddies = 0
            }
            return srvrMsg;
        }
        catch (IOException e)
        {
            log(ILogger.ASRT, "Buffer overflow in ServerMessage");
            return null;
        }
      }

    public BeginClientToServerAudioMessage(ServerMessage srvrMsg)
        throws java.io.IOException
    {
        super(srvrMsg.getHostAddress(), srvrMsg.getPort(),
              srvrMsg.getSequenceNumber(), srvrMsg.getSessionID(),
              srvrMsg.getType());

        DataInputStream in  = srvrMsg.getDataInput();
        m_groupsOrBuddies = (in.readByte() != 0);
        byte count = in.readByte();
        m_ids = new int[count];
        for (int i = 0; i < count; ++i)
        {
            m_ids[i] = in.readInt();
        }
    }

    public boolean getGroupsOrBuddies()
    {
        return m_groupsOrBuddies;
    }

    public int[] getIds()
    {
        return m_ids;
    }

    public void process(SessionMessageProcessor smp)
    {
        boolean fatalError = false;

        try
        {
            if (m_groupsOrBuddies == true)
            {
                log(ILogger.WARN, "processBeginAudioMessage request failed: "
                    + "Groups are not suppoted for BeginAudioMessage");
                fatalError = true;
            }
            else if( !smp.createVoiceSession())
            {
                log(ILogger.WARN, "processBeginAudioMessage request failed: "
                    + "audio session could not be created ");
                fatalError = true;
```

```
            }

            sendTextToClient(smp,
                BeginServerToClientAudioMessage.AUDIO_INDICATE);
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }

        try
        {
            // Create Recipient List
            Collection recipients = smp.createVoiceRecipientList(m_ids);
            Iterator recipientIterator = recipients.iterator();

            SessionMessageProcessor.VoiceRecipient recipient = null;
            m_authorId = (int) smp.getSubscriberID();
            m_voiceSessionId = (int) smp.getVoiceSessionID();
            while (recipientIterator.hasNext())
            {
                recipient = (SessionMessageProcessor.VoiceRecipient)
                    recipientIterator.next();

                if (recipient.is_online)
                {
                    BeginServerToClientAudioMessage msg =
                        new BeginServerToClientAudioMessage(
                        (int) recipient.sessionID, m_authorId, m_ids,
                        m_voiceSessionId);
                    smp.sendToOther(msg);
                    // System.err.println(msg);
                }
                else
                {
                    // TOCHECK: <CRW> 06-19-2002 SEND SMS to OFFLINE BUDDY
                    // ServerToClientTextMessage.sendTextMessage(smp,
                    //     "TODO send SMS to this offline client");
                }
            }
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }

    private void sendTextToClient(SessionMessageProcessor smp, String message)
    {
        try
        {
            ServerToClientTextMessage msg =
                new ServerToClientTextMessage((int) smp.getSessionID(),
                    (int) (int) smp.getSubscriberID(), m_ids, message);

            msg.process(smp);
            // System.err.println(msg);
            log(ILogger.TRCE,
                "Sent ServerToClientText message to author"
                + m_authorId);
        }
        catch (Exception e)
        {
            System.err.println("Error: Exception occured in: "
                + " BeginAudioMessage::sendTextToClient ");
            e.printStackTrace();
        }
```

```
        }
    }
```

```
*****************  Version 2  *****************
User: Mihamih        Date:  2/13/03  Time:  3:16p
Checked in $/FastText/Version1/client/versions/symbian/src
Comment:
  Fixed so play volume is set correctly. Added a couple of debug
messages.

*****************  Version 1  *****************
User: Mihamih        Date:  2/13/03  Time: 12:30a
Created AudioManager.cpp
Comment:

-----------------------------------------------------------------

#include "AudioManager.h"

#include <MdaAudioSampleEditor.h>
#include <mda/common/GsmAudio.h>
#include <eikconso.h>
#include <eikenv.h>
#include <aknnavide.h>
#include <stringloader.h>

#include "Context.h"
#include "Beeper.h"
#include "FastxtEngine.h"
#include "Connection.h"
#include "ClientMessage.h"
#include "RecepientQue.h"
#include "ReceivedMessagesView.h"
#include "fastxt.rsg"

// see RFC3267 and 3GPP TS 21.101 V4.1.0 (2001-06) for more info
// storage IDs for AMR Frame Types supported by 7650.
#define FRAME_DATA_5_15   0x0C // 5.15 Kbits
#define FRAME_DATA_7_40   0x24 // 7.40 Kbits TDMA-EFR
#define FRAME_DATA_12_2   0x3C // 12.2 Kbits GSM-EFR
#define FRAME_SID         0x44
#define FRAME_NO_DATA     0x7C

// frame lengths
#define FRAME_DATA_LEN_5_15 13
#define FRAME_DATA_LEN_7_40 19
#define FRAME_DATA_LEN_12_2 31
#define FRAME_SID_LEN        5
#define FRAME_NO_DATA_LEN    0

#define GSM_WAVE_HEADER_LEN  60
#define GSM_6_10_FRAME_LEN   325 // 200 ms 65 bytes per 40 ms.

#define FRAMES_TO_PACK 10 // 20 ms per frame for 200 ms.

static const TInt KAUDIO_PACKET_DATA_LEN =
    FRAMES_TO_PACK * (FRAME_DATA_LEN_12_2 + 1);

_LIT(KRecAudioFilesPath, "c:\\documents\\fastxt\\recorded\\");
_LIT(KPlayAudioFilesPath, "c:\\documents\\fastxt\\play\\");
_LIT(KRecordFileFormat, "rec%d.clp");
_LIT(KPlayFileFormat, "%S.clp");
_LIT8(KCLIP_MAGIC_AMR_SIGNATURE, "#!AMR\n");
_LIT8 (KCLIP_MAGIC_GSM_6_10,
      "RIFF....WAVEfmt .......................fact........data....");
```

Exhibit 2

```
_LIT(KAUDIO_START_TEXT, ")))");

CAudioManager::~CAudioManager()
{
    STACK_TRACE_ENTER_FUNCTION(_L("CAudioManager::~CAudioManager"));

    m_pRecorder->Close();
    m_pPlayer->Close();
    delete m_pRecorder;
    delete m_pPlayer;

    delete m_pAMRAudioType;
    delete m_pGSMAudioType;
    delete m_pRecorderIndicator;
    delete m_pRecordBuffer;
    delete m_pRecordPtr;

    CPlayIDNode* pNode;
    while (!m_playQueue.IsEmpty())
    {
        pNode = m_playQueue.First();
        m_playQueue.Remove(*pNode);
        delete pNode;
    }

    STACK_TRACE_EXIT_FUNCTION(_L("CAudioManager::~CAudioManager"));
}

void CAudioManager::ConstructL()
{
    STACK_TRACE_ENTER_FUNCTION(_L("CAudioManager::ConstructL"));

    m_playQueue.SetOffset(_FOFF(CPlayIDNode, m_link));

    m_pSettings = m_pContext->m_pEngine->GetSettings();

    m_pRecorderIndicator = m_pContext->m_pNaviPane->CreateVolumeIndicatorL(
        R_AVKON_NAVI_PANE_RECORDER_VOLUME_INDICATOR);

    m_pRecorder = CMdaAudioRecorderUtility::NewL(*this);

    m_pPlayer = CMdaAudioRecorderUtility::NewL(*this);

    if (m_pSettings->m_audioCodec == EGSM_6_10) // GSM 6.10
    {
        m_recordMonitor.ConstructL(this);
        m_pRecordBuffer = HBufC8::NewL(48750 + 60); // 30 secs + 60 b. header
        m_pRecordPtr = new (ELeave) TPtr8(m_pRecordBuffer->Des());

        m_recDesLocation.iDes = m_pRecordPtr;
    }
    else
    {
        SetFrameIDAndLength((TMdaRawAmrAudioCodec::TAmrMode)
            m_pSettings->m_audioCodec);
    }

    m_pAMRAudioType = new (ELeave) CMdaAudioType;
    m_pAMRAudioType->iFormat = new (ELeave) TMdaRawAmrClipFormat();
    m_pAMRAudioType->iCodec = new (ELeave) TMdaRawAmrAudioCodec(
        (TMdaRawAmrAudioCodec::TAmrMode)m_pSettings->m_audioCodec, ETrue);
    m_pAMRAudioType->iSettings = new (ELeave) TMdaAudioDataSettings;
    m_pAMRAudioType->iSettings->iSampleRate = 8000;
```

```
    m_pAMRAudioType->iSettings->iChannels = 1;

    m_pGSMAudioType = new (ELeave) CMdaAudioType;
    m_pGSMAudioType->iFormat = new (ELeave) TMdaWavClipFormat();
    m_pGSMAudioType->iCodec = new (ELeave) TMdaGsmWavCodec();
    m_pGSMAudioType->iSettings = new (ELeave) TMdaAudioDataSettings;
    m_pGSMAudioType->iSettings->iSampleRate = 8000;
    m_pGSMAudioType->iSettings->iChannels = 1;

    m_recState = ENone;
    m_playState = ENone;

    m_rfs = CEikonEnv::Static()->FsSession();

    CFileMan* pFileMan = CFileMan::NewL(m_rfs);
    CleanupStack::PushL(pFileMan);

    pFileMan->RmDir(KRecAudioFilesPath);
    CleanupStack::PopAndDestroy(); // pFileMan;
    User::LeaveIfError(m_rfs.MkDirAll(KRecAudioFilesPath));
    m_rfs.MkDirAll(KPlayAudioFilesPath);

    STACK_TRACE_EXIT_FUNCTION(_L("CAudioManager::ConstructL"));
}

bool CAudioManager::IsRecording()
{
    STACK_TRACE_ENTER_FUNCTION(_L("CAudioManager::IsRecording"));
    STACK_TRACE_EXIT_FUNCTION(_L("CAudioManager::IsRecording"));

    return m_recState == ERecording;
}

void CAudioManager::HandleCommandL(TInt /*aCommand*/)
{
    STACK_TRACE_ENTER_FUNCTION(_L("CAudioManager::HandleCommandL"));

    if (m_recState == ENone)
    {
        StartRecordingL();
    }
    else
    {
        StopRecordingL();
    }

    STACK_TRACE_EXIT_FUNCTION(_L("CAudioManager::HandleCommandL"));
}

void CAudioManager::StartRecordingL()
{
    STACK_TRACE_ENTER_FUNCTION(_L("CAudioManager::StartRecordingL"));

    if (m_recState == ENone)
    {
        if (m_pContext->m_pEngine->GetConnection()->IsConnected() == false)
        {
            Context::ShowErrorBoxL(R_STR_NO_CONNECTION);
        }
        else
        {
            if (m_playState != ENone)
            {
```

```
                m_pPlayer->Stop();
                m_pPlayer->Close();
                m_playState = ENone;
            }

            UpdateRecState(ERecording);

            m_pRecorder->Close();
            if (m_pSettings->m_audioCodec == EGSM_6_10)
            {
                m_pRecordPtr->Zero();
                m_bytesRead = GSM_WAVE_HEADER_LEN;

                m_pRecorder->OpenL(&m_recDesLocation, m_pGSMAudioType->iFormat,
                    m_pGSMAudioType->iCodec, m_pGSMAudioType->iSettings);
            }
            else
            {
                m_recFileLocation.iName.Copy(KRecAudioFilesPath);
                m_recFileLocation.iName.AppendFormat(KRecordFileFormat,
                    m_nextRecordClipId++);

                m_pRecorder->OpenL(&m_recFileLocation, m_pAMRAudioType->iFormat,
                    m_pAMRAudioType->iCodec, m_pAMRAudioType->iSettings);
            }
        }
    }
#if DEBUG_CONSOLE
    else
    {
        m_pContext->m_pConsole->Printf(_L(
            "Tried to record when already recording\n"));
    }
#endif

    STACK_TRACE_EXIT_FUNCTION(_L("CAudioManager::StartRecordingL"));
}

void CAudioManager::StopRecordingL()
{
    STACK_TRACE_ENTER_FUNCTION(_L("CAudioManager::StopRecordingL"));

    if (m_recState == ERecording)
    {
        UpdateRecState(ENone);
        m_pRecorder->Stop();
        if (m_pSettings->m_audioCodec == EGSM_6_10)
        {
            CheckRecordBufferL();
            SendEndAudioL();
        }
        else
        {
            SendClipL(m_recFileLocation.iName);
            m_rfs.Delete(m_recFileLocation.iName);
        }
    }
#if DEBUG_CONSOLE
    else
    {
        m_pContext->m_pConsole->Printf(_L(
            "Tried to stop recording when NOT recording\n"));
    }
```

```
#endif

    STACK_TRACE_EXIT_FUNCTION(_L("CAudioManager::StopRecordingL"));
}

void CAudioManager::MessageReceivedL(ClientMessage* pMsg)
{
    STACK_TRACE_ENTER_FUNCTION(_L("CAudioManager::MessageReceivedL"));

    int offset = ITransport::HEADER_LENGTH;
    int len;
    TUint8* pBytes = (TUint8*)pMsg->m_pBytes->Des().Ptr();

    switch(pMsg->GetType())
    {
    case ClientMessage::SERVER_START_AUDIO:
        HandleStartAudioL(pMsg);
        break;
    case ClientMessage::AUDIO:

        len = BigEndian::Get16(pBytes + offset);
        offset += 2;

        if (m_pReceivingClipID == NULL)
        {
#if DEBUG_CONSOLE
            m_pContext->m_pConsole->Printf(_L(
                "Got audio without audio start\n"));
#endif
            break;
        }

        if (m_receivingFileOpen == false)
        {
            TBuf<256> filename;
            TPtrC8 magic(NULL, 0);

            if (pBytes[offset] == EAMR)
            {
                m_currentPlayFileCodec = EAMR;
                magic.Set(KCLIP_MAGIC_AMR_SIGNATURE);
            }
            else if (pBytes[offset] == EGSM_6_10)
            {
                m_currentPlayFileCodec = EGSM_6_10;
                magic.Set(KCLIP_MAGIC_GSM_6_10);
            }
            else
            {
#if DEBUG_CONSOLE
                m_pContext->m_pConsole->Printf(_L("Bad codec\n"));
#endif
                break;
            }

            filename.Copy(KPlayAudioFilesPath);
            filename.AppendFormat(KPlayFileFormat, m_pReceivingClipID);

            TInt err = m_receivingClip.Create(m_rfs, filename, EFileWrite);
            if (err != KErrNone)
            {
#if DEBUG_CONSOLE
                m_pContext->m_pConsole->Printf(_L("Error %d opening file %S\n"),
```

```
                    err, &filename);
#endif
            }
            else
            {
                m_receivingFileOpen = true;
                WriteToClip(magic, 0, magic.Length());
            }
        }
        offset++;

        WriteToClip(*pMsg->m_pBytes, offset, len - 1);
        break;

    case ClientMessage::END_AUDIO:
        if (m_receivingFileOpen == true)
        {
            if (m_currentPlayFileCodec == EGSM_6_10)
            {
                // fill in the WAV header
                TBuf8<60> buf(KCLIP_MAGIC_GSM_6_10);
                TUint8* pBytes = (TUint8*)buf.Ptr();
                TInt size;

                m_receivingClip.Size(size);

                // File size - 8 (first 2 words)
                LittleEndian::Put32(pBytes + 4, size - 8);

                // fmt section size for GSM 6.10 is 20
                LittleEndian::Put32(pBytes + 16, 20);

                // audio format id for GSM 6.10 is 49
                LittleEndian::Put16(pBytes + 20, 49);

                // number of channels is 1
                LittleEndian::Put16(pBytes + 22, 1);

                // sample rate for GSM 6.10 is 8000
                LittleEndian::Put32(pBytes + 24, 8000);

                // byte rate for GSM 6.10 is 1625
                LittleEndian::Put32(pBytes + 28, 1625);

                // Block Align for GSM 6.10 is 65
                LittleEndian::Put32(pBytes + 32, 65);

                // Extra bytes is 2
                LittleEndian::Put16(pBytes + 36, 2);

                // I am not sure what this is but it seems constant
                LittleEndian::Put16(pBytes + 38, 320);

                // Fact section is always 4 bytes
                LittleEndian::Put32(pBytes + 44, 4);

                // this is the number of secons times the sample rate
                // which now seems to be correctly 8000
                // data size / 1625 * 8000
                size -= 60; // data size now
                LittleEndian::Put32(pBytes + 48, (TInt)size / 1625 * 8000);

                // finally the data size
```

```
            LittleEndian::Put32(pBytes + 56, size);

            m_receivingClip.Write(0, buf);
        }
        m_receivingClip.Close();

        CPlayIDNode* pNode = new (ELeave) CPlayIDNode();
        pNode->m_pPlayID = m_pReceivingClipID;
        m_playQueue.AddLast(*pNode);

        m_pReceivingClipID = NULL;
        PlayNextClipL();
    }
    break;
default:
    break;
}

    STACK_TRACE_EXIT_FUNCTION(_L("CAudioManager::MessageReceivedL"));
}

void CAudioManager::MoscoStateChangeEvent(CBase* aObject,
            TInt aPreviousState,TInt aCurrentState, TInt aErrorCode)
{
    STACK_TRACE_ENTER_FUNCTION(_L("CAudioManager::MoscoStateChangeEvent"));

#if DEBUG_CONSOLE
    m_pContext->m_pConsole->Printf(
        _L("aObject=%x, prev=%d, current=%d, error=%d\n"),
        aObject, aPreviousState, aCurrentState, aErrorCode);
#endif

    if (aObject == m_pRecorder)
    {
        if (aErrorCode != KErrNone)
        {
            m_pRecorder->Close();
            UpdateRecState(ENone);
        }
        else
        {
            if (aCurrentState == CMdaAudioRecorderUtility::EOpen)
            {
                TInt gain = m_pSettings->m_recordVolume;
                m_pRecorder->SetGain(m_pRecorder->MaxGain() * gain / 10);




                CAknVolumeControl* pCtrl = (CAknVolumeControl*)
                    m_pRecorderIndicator->DecoratedControl();
                pCtrl->SetValue(gain);

                m_pContext->m_pNaviPane->PushL(*m_pRecorderIndicator);
                m_pRecorder->RecordL();
                if (m_pSettings->m_audioCodec == EGSM_6_10)
                {
                    SendStartAudioL();
                    m_seq = 1;
                    m_recordMonitor.Start(1000000, 1000000, TCallBack(
                        CAudioManager::CRecMonitor::MonitorRunL, this));
                }
```

```
            }
        }
    }
    else if (aObject == m_pPlayer)
    {
        if (aErrorCode == KErrNone)
        {
            if (m_playState == EPlay)
            {
                m_pPlayer->SetVolume(m_pPlayer->MaxVolume() * 10 /
                    m_pSettings->m_speakerVolume);





                m_pPlayer->PlayL();
                m_playState = EPlaying;
            }
            else if (aCurrentState == CMdaAudioRecorderUtility::EOpen)
            {
                m_pPlayer->Close();
                m_playState = ENone;
            }
        }
        else
        {
            m_pPlayer->Close();
            m_playState = ENone;
        }
    }

    if (m_playState == ENone && m_recState == ENone &&
        !m_playQueue.IsEmpty())
    {
        PlayNextClipL();
    }

    STACK_TRACE_EXIT_FUNCTION(_L("CAudioManager::MoscoStateChangeEvent"));
}

void CAudioManager::SendClipL(const TDesC& clipName)
{
    STACK_TRACE_ENTER_FUNCTION(_L("CAudioManager::SendClipL"));

    SendStartAudioL();
    SendAudioDataL(clipName);
    SendEndAudioL();

    STACK_TRACE_EXIT_FUNCTION(_L("CAudioManager::SendClipL"));
}

void CAudioManager::SendAudioDataL(const TDesC& clipName)
{
    STACK_TRACE_ENTER_FUNCTION(_L("CAudioManager::SendAudioDataL"));

    RFile clipFile;
    TInt i, frameLen, seq = 1;
    ClientMessage* pMsg = NULL;
    TUint8* pBytes;
    TBuf8<FRAME_DATA_LEN_12_2> amrBuf;
    bool hasError = false;
```

```
    m_pRecorder->Close();
    TInt err = clipFile.Open(m_rfs, clipName, EFileRead);
    if (err != KErrNone)
    {
        TBuf<64> buf;
        buf.AppendFormat(
            StringLoader::LoadLC(R_STR_COULD_NOT_OPEN_CLIP)->Des(), err);
        CleanupStack::PopAndDestroy(); // StringLoader::LoadLC()

        Context::ShowErrorBoxL(buf);
        hasError = true;
    }

    if (hasError == false)
    {
        TPtrC8 magic(KCLIP_MAGIC_AMR_SIGNATURE);
        clipFile.Read(amrBuf, magic.Length());
        if (amrBuf.Compare(magic) != 0)
        {
            Context::ShowErrorBoxL(R_STR_BAD_CLIP_FILE);
            hasError = true;
        }
    }

    if (hasError == false)
    {
        while(&i != NULL) // while(true) without a warning
        {
            pMsg = new (ELeave) ClientMessage(ClientMessage::AUDIO,
                3 + KAUDIO_PACKET_DATA_LEN);
            CleanupStack::PushL(pMsg);
            pMsg->ConstructL();
            TPtr8 des = pMsg->m_pBytes->Des();
            pBytes = (TUint8*)des.Ptr();

            des.SetLength(ITransport::HEADER_LENGTH + 3);
            for (i = 0; i < FRAMES_TO_PACK; i++)
            {
                User::LeaveIfError(clipFile.Read(amrBuf, 1));
                if (amrBuf.Length() == 0)
                {
                    break;
                }
                des.Append(amrBuf);

                if (amrBuf[0] == m_frameDataID)
                {
                    frameLen = m_frameDataLength;
                }
                else if (amrBuf[0] == FRAME_SID)
                {
                    frameLen = FRAME_SID_LEN;
                }
                else if (amrBuf[0] == FRAME_NO_DATA)
                {
                    frameLen = FRAME_NO_DATA_LEN;
                }
                else
                {
                    frameLen = -1;
#if DEBUG_CONSOLE
                    m_pContext->m_pConsole->Printf(
```

```
                    _L("Unknown frame type 0x%X\n"), amrBuf[0]);
#endif
                }

                if (frameLen > 0)
                {
                    User::LeaveIfError(clipFile.Read(amrBuf, frameLen));
                    if (amrBuf.Length() < frameLen)
                    {
                        // roll back the frame id byte
                        des.SetLength(des.Length() - 1);
                        break;
                    }
                    des.Append(amrBuf);
                }
                else if (frameLen == -1)
                {
                    hasError = true;
                    break;
                }
            }

            pMsg->m_seq = seq++;
            pMsg->m_len = des.Length() - ITransport::HEADER_LENGTH;
            BigEndian::Put16(pBytes + ITransport::HEADER_LENGTH,
                (TUint16)(pMsg->m_len - 2));

            if (m_pSettings->m_audioCodec == EGSM_6_10)
            {
                pBytes[ITransport::HEADER_LENGTH + 2] = EGSM_6_10;
            }
            else
            {
                pBytes[ITransport::HEADER_LENGTH + 2] = EAMR;
            }

            if (pMsg->m_len == ITransport::HEADER_LENGTH + 2)
            {
                // nothing was packed in this frame
                CleanupStack::PopAndDestroy();
            }
            else
            {
                CleanupStack::Pop();
                m_pContext->m_pEngine->SendMessageL(pMsg);
            }

            if (i < FRAMES_TO_PACK)
    {
                break;
            }
    }
    }

    clipFile.Close();

    STACK_TRACE_EXIT_FUNCTION(_L("CAudioManager::SendAudioDataL"));
}

void CAudioManager::WriteToClip(const TDesC8& desc, TInt offset, TInt len)
{
    STACK_TRACE_ENTER_FUNCTION(_L("CAudioManager::WriteToClip"));
```

```
    TUint8* pBytes = (TUint8*)desc.Ptr();
    TPtr8 ptr(pBytes + offset, len, len);
    TInt err = m_receivingClip.Write(ptr);

#if DEBUG_CONSOLE
    if (err != KErrNone)
    {
        m_pContext->m_pConsole->Printf(
            _L("Error %d while writing clip\n"), err);
    }
#endif

    STACK_TRACE_EXIT_FUNCTION(_L("CAudioManager::WriteToClip"));
}

void CAudioManager::HandleStartAudioL(ClientMessage* pMsg)
{
    STACK_TRACE_ENTER_FUNCTION(_L("CAudioManager::HandleStartAudioL"));

    HBufC16* pAuthor;
    int i, offset, len, size;
    CBuddyInfo* pBuddyInfo;
    TRecepientQue* pRecepientQue;
    TUint8* pBytes = (TUint8*)pMsg->m_pBytes->Des().Ptr();
    bool echoError = false;

    pRecepientQue = new (ELeave) TRecepientQue(true);
    CleanupStack::PushL(pRecepientQue);
    pRecepientQue->SetOffset(_FOFF(CBuddyInfo,iLink));

    pBuddyInfo = new (ELeave) CBuddyInfo(true, true);
    CleanupStack::PushL(pBuddyInfo);

    offset = ITransport::HEADER_LENGTH;
    pBuddyInfo->m_id = BigEndian::Get32(pBytes + offset);
    offset += 4;

    len = BigEndian::Get16(pBytes + offset);
    offset += 2;

    pAuthor = m_pContext->FromUtf(pBytes + offset, len)->AllocL();
    pBuddyInfo->m_pName = pAuthor;
    offset += len;

    pBuddyInfo->m_pPhone = HBufC::NewL(0);

    pRecepientQue->AddLast(*pBuddyInfo);
    CleanupStack::Pop(); // buddyInfo

    size = pBytes[offset++];
    for (i = 0; i < size; i++)
    {
        pBuddyInfo = new (ELeave) CBuddyInfo(true, true);
        CleanupStack::PushL(pBuddyInfo);

        pBuddyInfo->m_id = BigEndian::Get32(pBytes + offset);
        offset += 4;

        len = BigEndian::Get16(pBytes + offset);
        offset += 2;

        pBuddyInfo->m_pName = m_pContext->FromUtf(
            pBytes + offset, len)->AllocL();
```

```
        offset += len;

        if (pBuddyInfo->m_pName->Left(1).Compare(_L("!")) == 0)
        {
            echoError = true;
        }

        pBuddyInfo->m_pPhone = HBufC::NewL(0);

        pRecepientQue->AddLast(*pBuddyInfo);
        CleanupStack::Pop(); // pBuddyInfo;
    }

    NewReceivedMessageL();
    CleanupStack::PushL(m_pReceivingClipID);
    m_pContext->m_pReceivedMessagesView->AddNewItemL(pRecepientQue,
        (&KAUDIO_START_TEXT)->AllocLC(), m_pContext->m_pReceivedMessagesView->
        PickIcon(pRecepientQue, echoError), m_pReceivingClipID);
    CleanupStack::Pop(); // m_pReceivingClipID

    CleanupStack::Pop(2); // pMstText, pRecepientQueue

    STACK_TRACE_EXIT_FUNCTION(_L("CAudioManager::HandleStartAudioL"));
}

void CAudioManager::PlayNextClipL()
{
    STACK_TRACE_ENTER_FUNCTION(_L("CAudioManager::PlayNextClipL"));

    if (m_playState == ENone && m_recState == ENone &&
        m_playQueue.IsEmpty() == EFalse)
    {
        CPlayIDNode* pNode = m_playQueue.First();
        m_playQueue.Remove(*pNode);
        CleanupStack::PushL(pNode);

        PlayClipL(pNode->m_pPlayID);

        CleanupStack::PopAndDestroy(); // pNode;
    }

    STACK_TRACE_EXIT_FUNCTION(_L("CAudioManager::PlayNextClipL"));
}

void CAudioManager::SetFrameIDAndLength(TMdaRawAmrAudioCodec::TAmrMode amrMode)
{
    STACK_TRACE_ENTER_FUNCTION(_L("CAudioManager::SetFrameIDAndLength"));

    switch(amrMode)
    {
    case TMdaRawAmrAudioCodec::EMR515:
        m_frameDataID = FRAME_DATA_5_15;
        m_frameDataLength = FRAME_DATA_LEN_5_15;
        break;
    case TMdaRawAmrAudioCodec::EMR74:
        m_frameDataID = FRAME_DATA_7_40;
        m_frameDataLength = FRAME_DATA_LEN_7_40;
        break;
    case TMdaRawAmrAudioCodec::EMR122:
        m_frameDataID = FRAME_DATA_12_2;
        m_frameDataLength = FRAME_DATA_LEN_12_2;
        break;
    default:
```

```
#if DEBUG_CONSOLE
        m_pContext->m_pConsole->Printf(_L("Unknown audio codec\n"));
#endif
        break;
    }

    STACK_TRACE_EXIT_FUNCTION(_L("CAudioManager::SetFrameIDAndLength"));
}

void CAudioManager::UpdateRecState(EAudioState recState)
{
    STACK_TRACE_ENTER_FUNCTION(_L("CAudioManager::UpdateRecState"));

    m_recState = recState;
    if (recState != ERecording)
    {
        m_pContext->m_pNaviPane->Pop(m_pRecorderIndicator);
        m_recordMonitor.Cancel();
    }

    STACK_TRACE_EXIT_FUNCTION(_L("CAudioManager::UpdateRecState"));
}

void CAudioManager::CheckRecordBufferL()
{
    STACK_TRACE_ENTER_FUNCTION(_L("CAudioManager::CheckRecordBufferL"))

    ClientMessage* pMsg;
    TUint8 *pSrcBytes, *pDstBytes;

    pSrcBytes = (TUint8*)m_pRecordBuffer->Des().Ptr();

    while (m_pRecordBuffer->Length() - m_bytesRead >= GSM_6_10_FRAME_LEN)
    {
        pMsg = new (ELeave) ClientMessage(ClientMessage::AUDIO,
                3 + GSM_6_10_FRAME_LEN);
        CleanupStack::PushL(pMsg);
        pMsg->ConstructL();
        pMsg->m_seq = m_seq++;
        pDstBytes = (TUint8*)pMsg->m_pBytes->Des().Ptr();

        Mem::Copy(pDstBytes + ITransport::HEADER_LENGTH + 3,
            pSrcBytes + m_bytesRead, GSM_6_10_FRAME_LEN);
        m_bytesRead += GSM_6_10_FRAME_LEN;

        BigEndian::Put16(pDstBytes + + ITransport::HEADER_LENGTH,
            (TUint16)(pMsg->m_len - 2));
        pDstBytes[ITransport::HEADER_LENGTH + 2] = EGSM_6_10;

        CleanupStack::Pop();
        m_pContext->m_pEngine->SendMessageL(pMsg);
    }

    if (m_recState == ENone &&
        m_pRecordBuffer->Length() - m_bytesRead > 0)
    {
        TInt len = m_pRecordBuffer->Length() - m_bytesRead;
        pMsg = new (ELeave) ClientMessage(ClientMessage::AUDIO,
            3 + len);
        CleanupStack::PushL(pMsg);
        pMsg->ConstructL();
        pMsg->m_seq = m_seq++;
        pDstBytes = (TUint8*)pMsg->m_pBytes->Des().Ptr();
```

```
        Mem::Copy(pDstBytes + ITransport::HEADER_LENGTH + 3,
            pSrcBytes + m_bytesRead, len);
        m_bytesRead += len;

        BigEndian::Put16(pDstBytes + ITransport::HEADER_LENGTH,
            (TUint16)(pMsg->m_len - 2));
        pSrcBytes[ITransport::HEADER_LENGTH + 2] = EGSM_6_10;

        CleanupStack::Pop();
        m_pContext->m_pEngine->SendMessageL(pMsg);
    }

    STACK_TRACE_EXIT_FUNCTION(_L("CAudioManager::CheckRecordBufferL"))
}

void CAudioManager::SendStartAudioL()
{
    STACK_TRACE_ENTER_FUNCTION(_L("CAudioManager::SendStartAudioL"))

    int offset;
    TUint8 len = 0;
    CBuddyInfo* pRecip;

    ClientMessage* pMsg = new (ELeave) ClientMessage(
        ClientMessage::CLIENT_START_AUDIO, 2 +
        m_pContext->m_pRecepientQue->Size() * 4);
    CleanupStack::PushL(pMsg);
    pMsg->ConstructL();

    TUint8* pBytes = (TUint8*)pMsg->m_pBytes->Des().Ptr();

    offset = ITransport::HEADER_LENGTH;
    pBytes[offset++] = 0;        // send to buddies
    offset++; // we fill in the length later

    TSglQueIter<CBuddyInfo> itr(*m_pContext->m_pRecepientQue);
    while ((pRecip = itr++) != NULL)
    {
        if (pRecip->m_id != 0)
        {
            len++;
            BigEndian::Put32(pBytes + offset, pRecip->m_id);
            offset += 4;

            if (m_pContext->m_replyToAll == false)
            {
                break;
            }
        }
    }
    pBytes[ITransport::HEADER_LENGTH + 1] = len;

    CleanupStack::Pop();
    m_pContext->m_pEngine->SendMessageL(pMsg);

    STACK_TRACE_EXIT_FUNCTION(_L("CAudioManager::SendStartAudioL"))
}

void CAudioManager::SendEndAudioL()
{
    STACK_TRACE_ENTER_FUNCTION(_L("CAudioManager::SendEndAudioL"))
```

```
    ClientMessage *pMsg;

    pMsg = new (ELeave) ClientMessage(
        ClientMessage::END_AUDIO, 0);
    CleanupStack::PushL(pMsg);
    pMsg->ConstructL();
    CleanupStack::Pop(); // pMsg
    m_pContext->m_pEngine->SendMessageL(pMsg);

    STACK_TRACE_EXIT_FUNCTION(_L("CAudioManager::SendEndAudioL"))
}

CAudioManager::EAudioCodec CAudioManager::GetAudioCodec(TDesC& name)
{
    STACK_TRACE_ENTER_FUNCTION(_L("CAudioManager::GetAudioCodec"))

    RFile file;
    EAudioCodec codec = EAMR;

    TInt err = file.Open(m_rfs, name, EFileRead);
    if (err == KErrNone)
    {
        TBuf8<1> buf;
        file.Read(buf);
        if (buf[0] == 'R')
        {
            codec = EGSM_6_10;
        }
        file.Close();
    }

    STACK_TRACE_EXIT_FUNCTION(_L("CAudioManager::GetAudioCodec"))

    return codec;
}

void CAudioManager::NewReceivedMessageL()
{
    STACK_TRACE_ENTER_FUNCTION(_L("CAudioManager::NewReceivedMessageL"))

    TBuf<64> fileID;
    TTime time;
    TDateTime dateTime;

    time.HomeTime();
    dateTime = time.DateTime();

    if (m_pReceivingClipID != NULL)
    {
#if DEBUG_CONSOLE
        m_pContext->m_pConsole->Printf(_L("Warning: Audio start with no end\n"));
#endif
        m_receivingClip.Close();

        CPlayIDNode* pNode = new (ELeave) CPlayIDNode();
        pNode->m_pPlayID = m_pReceivingClipID;
        m_playQueue.AddLast(*pNode);

        m_pReceivingClipID = NULL;
        PlayNextClipL();
    }

    fileID.AppendFormat(_L("%d_%d_%d_%d_%d_%d_%d"), dateTime.Year(),
```

```
        dateTime.Month(), dateTime.Day(), dateTime.Hour(), dateTime.Minute(),
        dateTime.Second(), dateTime.MicroSecond());

    m_pReceivingClipID = fileID.AllocL();
    m_receivingFileOpen = false;

    STACK_TRACE_EXIT_FUNCTION(_L("CAudioManager::NewReceivedMessageL"))
}

void CAudioManager::PlayClipL(TDesC* pClipID)
{
    STACK_TRACE_ENTER_FUNCTION(_L("CAudioManager::PlayClipL"))

    if (m_playState != ENone || m_recState != ENone)
    {
        Context::ShowErrorBoxL(R_STR_BUSY);
        return;
    }

    m_playState = EPlay;
    m_playFileLocation.iName.Copy(KPlayAudioFilesPath);
    m_playFileLocation.iName.AppendFormat(KPlayFileFormat, pClipID);
    EAudioCodec codec = GetAudioCodec(m_playFileLocation.iName);
    if (codec == EGSM_6_10)
    {
        m_pPlayer->OpenL(&m_playFileLocation, m_pGSMAudioType->iFormat,
            m_pGSMAudioType->iCodec, m_pGSMAudioType->iSettings);
    }
    else
    {
        m_pPlayer->OpenL(&m_playFileLocation, m_pAMRAudioType->iFormat,
            m_pAMRAudioType->iCodec, m_pAMRAudioType->iSettings);
    }
#if DEBUG_CONSOLE
    m_pContext->m_pConsole->Printf(_L("Playing clip %S\n"), pClipID);
#endif

    STACK_TRACE_EXIT_FUNCTION(_L("CAudioManager::PlayClipL"))
}

void CAudioManager::DeletePlayClipL(TDesC* pClipID)
{
    STACK_TRACE_ENTER_FUNCTION(_L("CAudioManager::DeletePlayClipL"))

    TBuf<256> clipFile;

    clipFile.Copy(KPlayAudioFilesPath);
    clipFile.AppendFormat(KPlayFileFormat, pClipID);
    m_rfs.Delete(clipFile);

    STACK_TRACE_EXIT_FUNCTION(_L("CAudioManager::DeletePlayClipL"))
}

void CAudioManager::DeleteAllPlayClipsL()
{
    STACK_TRACE_ENTER_FUNCTION(_L("CAudioManager::DeleteAllPlayClipsL"))

    CFileMan* pFileMan = CFileMan::NewL(m_rfs);
    CleanupStack::PushL(pFileMan);
    pFileMan->RmDir(KPlayAudioFilesPath);
    CleanupStack::PopAndDestroy(); // pFileMan;
    m_rfs.MkDirAll(KPlayAudioFilesPath);
```

```
    STACK_TRACE_EXIT_FUNCTION(_L("CAudioManager::DeleteAllPlayClipsL"))
}
```

```
$/v1_client/client/kth_demo/src_core/AudioManager.cpp

*****************  Version 2  *****************
User: Chris Wilson    Date: 10/18/02  Time: 12:37p
Checked in $/Version1/client/kth_demo/src_core
Comment:
  Added CE support

*****************  Version 1  *****************
User: Chris Wilson    Date: 10/16/02  Time: 12:00p
Created AudioManager.cpp
Comment:
  Rearranged Source files for easier CE/Win32 Interop

---------------------------------------------------------------

/*

::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::
  :: YoMobile Inc.
  ::
  :: Copyright (c) 2002, YoMobile Inc. All rights reserved.
  ::
  :: Use, distribution, or disclosure of this content in full or in part
is
  :: prohibited without prior written authorization from YoMobile Inc.
  ::

::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::
*/
#include "stdafx.h"
#include <afxmt.h>
// #include <process.h>    /* _beginthread, _endthread */
#include "Thread.h"
#include "AudioManager.h"
#include "AudioMessage.h"
#include "MessageManager.h"
#include "EndAudioMessage.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

#define VERIFY_VALS false

int AudioManager::m_recIdx = 0;
int AudioManager::m_clientsRecording = 0;
static int lastRxAudioSeq = 0;

CList<WAVEHDR*, WAVEHDR*> AudioManager::m_playWaveHeaders;
CList<AudioStream*, AudioStream*> playList;
CMutex m_streamMutex;
CSingleLock slock(&m_streamMutex);
CMutex m_playListMutex;
CSingleLock sPlaylock(&m_playListMutex);

static void playNonStream(void* pAudioManager, AudioStream* stream);
```

Exhibit
3

```
static void playStream(void* pAudioManager, AudioStream* stream);
static unsigned int transferAudioOut(void* pAudioManager);
static unsigned int playAudioStreams(void* pAudioManager);
void DoEvents1();
bool AudioManager::m_shuttingDown = true;

MessageManager* pMessageManager;
bool g_outThreadRunning = false;

AudioManager::AudioManager(void* messageManager)
{
      ASSERT(messageManager != NULL);
      pMessageManager = (MessageManager*) messageManager;

      m_pAudioOutThread     = new YoMobile::Thread(transferAudioOut,
                                                   this);
      m_pPlayAudioThread     = new YoMobile::Thread(playAudioStreams,
                                                   this);
      ASSERT(m_pAudioOutThread != NULL);
      ASSERT(m_pPlayAudioThread != NULL);

      lastRxAudioSeq = 0;
      m_txAudioSeq = 0;
      m_iterationLock = false;
      m_pClientsList = new ClientNode();
      m_pClientsList->m_encoding = ENCODING_G723;
      m_isPlaying = false;
      m_hWaveOut = NULL;
      m_hWaveIn = NULL;

      m_pAudioCollection = new AudioCollection();

      // Initialize G723
    WrkRate = Rate53;
    Init_Coder();
    Init_Decod();

      if (m_lastAudioFrameBuf.m_pData == NULL)
      {
            m_lastAudioFrameBuf.m_pData = new char[G723_BUFFER_LENGTH];
      }

    memset(m_lastAudioFrameBuf.m_pData, 0, G723_BUFFER_LENGTH);
    m_lastAudioFrameBuf.m_len = G723_BUFFER_LENGTH;
      // make sure no one tries to delete it.
      m_lastAudicFrameBuf.m_refCount = 1;


      openAudioOut();

      for (int i = 0; i < REC_BUFFER_NR; i++)
    {
        memset(&m_recHdr[i], 0, sizeof(WAVEHDR));
        m_recHdr[i].lpData = new char[BUFFER_LENGTH];
        m_recHdr[i].dwBufferLength = BUFFER_LENGTH;
    }
      m_shuttingDown = false;

      m_pPlayAudioThread->Run();
}
```

```
AudioManager::~AudioManager()
{
      MYTRACE0("Destroying Audion Manager!");
      int i=0;

      // Make sure no recording
      ASSERT(m_pClientsList != NULL);
      m_pClientsList->m_recording = false;

      stopRecording();
      m_clientsRecording = 0;

    if (m_pAudioCollection != NULL)
    {
        delete m_pAudioCollection;
        m_pAudioCollection = NULL;
    }

      closeWaveIn();

      closeAudioOut();

      clearAllAudioFrames();

    if (m_pAudioOutThread != NULL)
    {
        delete m_pAudioOutThread;
        m_pAudioOutThread = NULL;
    }

    if (m_pPlayAudioThread != NULL)
    {
        delete m_pPlayAudioThread;
        m_pPlayAudioThread = NULL;
    }

    // Wait for playback or record to finish
      while (m_iterationLock && !m_shuttingDown)
      {
            Sleep(10);
      }

      for (i = 0; i < REC_BUFFER_NR; i++)
    {
        delete [] m_recHdr[i].lpData;
    }

      if (m_pClientsList != NULL)
      {
            delete m_pClientsList;
            m_pClientsList = NULL;
      }

    if (m_lastAudioFrameBuf.m_pData != NULL)
    {
        delete[] m_lastAudioFrameBuf.m_pData;
        m_lastAudioFrameBuf.m_pData = NULL;
    }
}

//
```

```
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::
// ::
// :: Function name: AudioManager::reclaimPlayWaveHeaders()
// ::
// :: Function Parameters: NONE
// ::
// :: Function Return: bool
// ::       (true) headers were reclaimed
// ::       (false) unable to reclaim header
// ::
// :: Funtion Purpose: cleans up the preparation performed
// ::                         by the decodeAndPlay
// ::
// :: Notes:
// ::
// ::
// ::
//
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::
bool AudioManager::reclaimPlayWaveHeaders()
{
      POSITION pos;
    WAVEHDR* pWaveHdr;
    bool res = false;

    pos = m_playWaveHeaders.GetHeadPosition();
    if (pos != NULL)
    {
        pWaveHdr = m_playWaveHeaders.GetAt(pos);
        if ((pWaveHdr->dwFlags & WHDR_DONE) != 0)
        {
            checkResult("waveOutUnprepareHeader", waveOutUnprepareHeader
(
                m_hWaveOut, pWaveHdr, sizeof(WAVEHDR)));
            res = true;
            m_playWaveHeaders.RemoveAt(pos);
            delete [] pWaveHdr->lpData;
            delete pWaveHdr;
        }
    }

    return res;
}

//
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::
// ::
// :: Function name: AudioManager::stopRecording()
// ::
// :: Function Parameters: NONE
// ::
// :: Function Return: NONE
// ::
// :: Funtion Purpose: stops waveform-audio input.
// ::
// :: Notes:
// ::
// ::
```

```
// ::
//
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::
void AudioManager::stopRecording()
{
      // MYTRACE0("AudioMgr: stopRecording()\r\n");
      Sleep(200);
      ASSERT(m_pClientsList != NULL);
      if (m_pClientsList->m_recording == false)
    {
        return;
    }

      m_clientsRecording = 0;

      m_lastAudioFrameBuf.m_refCount++;
    m_pClientsList->m_sendBufferList.AddTail(
            new BufferNode(&m_lastAudioFrameBuf));

      closeWaveIn();
      m_pClientsList->m_recording = false;
}

//
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::
// ::
// :: Function name: AudioManager::closeWaveIn()
// ::
// :: Function Parameters: NONE
// ::
// :: Function Return: NONE
// ::
// :: Funtion Purpose: stops waveform-audio input.
// ::
// :: Notes:
// ::
// ::
// ::
//
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::
void AudioManager::closeWaveIn()
{
      MYTRACE0("Closing Wave In");
      if (m_hWaveIn != NULL)
    {
        checkResult("waveInStop", waveInStop(m_hWaveIn));
        checkResult("waveInReset", waveInReset(m_hWaveIn));

            checkRecordWaveHeaders();
            ASSERT(m_pClientsList != NULL);
//MKFIX0312       m_pClientsList->m_recording = false;

        checkResult("waveInClose", waveInClose(m_hWaveIn));
    }

    m_hWaveIn = NULL;
}
```

```
//
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::
// ::
// :: Function name: AudioManager::startRecording()
// ::
// :: Function Parameters: NONE
// ::
// :: Function Return: NONE
// ::
// :: Funtion Purpose: opens the given waveform-audio input device for
recording.
// ::
// :: Notes:
// ::
// ::
// ::
//
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::
void AudioManager::startRecording()
{
      ASSERT(m_pClientsList != NULL);
      ASSERT(m_pClientsList->m_recording == false);

    while (g_outThreadRunning == true)
    {
        Sleep(100);
    }

    m_clientsRecording++;
    m_pClientsList->m_recording = true;

      VERIFY(m_hWaveIn == NULL);

#if defined(_WIN32_WCE)
    closeAudioOut();
#endif

    WAVEFORMATEX wfx;
    MMRESULT mmresult;

    wfx.cbSize = 0;
    wfx.nBlockAlign = 2;
    wfx.nAvgBytesPerSec = 8000 * wfx.nBlockAlign;
    wfx.nChannels = 1;
    wfx.nSamplesPerSec = 8000;
    wfx.wBitsPerSample = 16;
    wfx.wFormatTag = WAVE_FORMAT_PCM;

    mmresult = waveInOpen(
                    &m_hWaveIn,
                    WAVE_MAPPER,
                    &wfx,
                    NULL,
                    NULL,
                    CALLBACK_NULL );

    checkResult("waveInOpen", mmresult);

    g_outThreadRunning = true;
```

```
        m_pAudioOutThread->Run();
}

void AudioManager::resetAudioSeq()
{
        m_txAudioSeq = 0;
}

//
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::
// ::
// :: Function name: AudioManager::isRecording()
// ::
// :: Function Parameters: NONE
// ::
// :: Function Return:
// ::         (TRUE) - if client is recording
// ::         (FALSE) - if client is not recording
// ::
// :: Funtion Purpose:
// ::
// :: Notes:
// ::
// ::
// ::
//
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::
bool AudioManager::isRecording()
{
        ASSERT(m_pClientsList != NULL);
        return m_pClientsList->m_recording;
}

//
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::
// ::
// :: Function name: AudioManager::encodeAndQueue()
// ::
// :: Function Parameters:
// ::         pData -
// ::         len -
// ::
// :: Function Return: NONE
// ::
// :: Funtion Purpose:
// ::
// :: Notes:
// ::
// ::
// ::
//
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::
void AudioManager::encodeAndQueue(char* pData, int len)
{
        ASSERT(m_pClientsList != NULL);

        WrkRate = Rate53;
```

```
        SharedBufferNode* pSharedUlawNode = NULL;
    SharedBufferNode* pSharedG723Node = NULL;

        ClientNode* pClientNode = m_pClientsList;
    int i, decodOffset, encodeOffset;


    if (pClientNode->m_encoding == ENCODING_MULAW)
    {
        if (pSharedUlawNode == NULL)
        {
            len = len / 2;
            VERIFY (len == MULAW_BUFFER_LENGTH);
            pSharedUlawNode = new SharedBufferNode();
            pSharedUlawNode->m_len = len;
            pSharedUlawNode->m_pData = new char[len];

            for (i = 0; i < len; i++)
            {
                pSharedUlawNode->m_pData[i] = linear2ulaw(
                        ((short*)pData)[i]);
            }
        }

        pSharedUlawNode->m_refCount++;
        pClientNode->m_sendBufferList.AddTail(
            new BufferNode(pSharedUlawNode));
    }
    else if (pClientNode->m_encoding == ENCODING_G723)
    {
        if (pSharedG723Node == NULL)
        {
            len = G723_BUFFER_LENGTH;

            pSharedG723Node = new SharedBufferNode();
            pSharedG723Node->m_len = len;
            pSharedG723Node->m_pData = new char[len + 4];

            decodOffset = encodeOffset = 0;
            int lineSize= 0;
                for (i = 0; i < G723_FRAMES; i++)
            {
#if defined(_WIN32_WCE)
                Coder((Word16*)(pData + decodOffset),
                    pSharedG723Node->m_pData + encodeOffset);
#else
                Read_lbc_buf(m_pG723DataBuf, Frame, Frame,
                    (Word16*)(pData + decodOffset));

                Coder(m_pG723DataBuf,
                        pSharedG723Node->m_pData + encodeOffset);
#endif

                int lineSize = GetLineSize(pSharedG723Node->m_pData +
                    encodeOffset);

                        if (VERIFY_VALS)
                            VERIFY(lineSize == 20);

                decodOffset += 480;
```

```
                encodeOffset += 20;

            }

            pSharedG723Node->m_refCount++;
            pClientNode->m_sendBufferList.AddTail(
                new BufferNode(pSharedG723Node));
        }
    }
    else
    {
        VERIFY(!"Unknown encoding!");
    }
}

//
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::
// ::
// :: Function name: AudioManager::checkRecordWaveHeaders()
// ::
// :: Function Parameters: NONE
// ::
// :: Function Return: bool
// ::       (true) -
// ::       (false) -
// ::
// :: Funtion Purpose:
// ::
// :: Notes:
// ::
// ::
// ::
//
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::
bool AudioManager::checkRecordWaveHeaders()
{

    bool res = false;

    if (m_hWaveIn == NULL)
    {
        // hold until recording starts/resumes
        return res;
    }

    while ((m_recHdr[m_recIdx].dwFlags & WHDR_DONE) != 0 ||
         (m_clientsRecording && m_recHdr[m_recIdx].dwFlags == 0))
    {
        res = true;


        if ((m_recHdr[m_recIdx].dwFlags & WHDR_DONE) != 0)
        {
            if (m_recHdr[m_recIdx].dwBytesRecorded > 0 &&
                m_recHdr[m_recIdx].dwBytesRecorded <
                m_recHdr[m_recIdx].dwBufferLength)
            {
                memset(m_recHdr[m_recIdx].lpData +
                    m_recHdr[m_recIdx].dwBytesRecorded, 0,
```

```
                    m_recHdr[m_recIdx].dwBufferLength -
                    m_recHdr[m_recIdx].dwBytesRecorded);

                m_recHdr[m_recIdx].dwBytesRecorded =
                    m_recHdr[m_recIdx].dwBufferLength;
            }

            if(m_recHdr[m_recIdx].dwBytesRecorded ==
                m_recHdr[m_recIdx].dwBufferLength)
            {
                encodeAndQueue(m_recHdr[m_recIdx].lpData,
                    m_recHdr[m_recIdx].dwBufferLength);

            }

            checkResult("waveInUnprepareHeader",
                waveInUnprepareHeader(m_hWaveIn, &m_recHdr[m_recIdx],
                                        sizeof(WAVEHDR)));

            m_recHdr[m_recIdx].dwFlags = 0;
        }

        if (m_clientsRecording > 0)
        {
            checkResult("waveInPrepareHeader", waveInPrepareHeader
(m_hWaveIn,
                &m_recHdr[m_recIdx], sizeof(WAVEHDR)));
            checkResult("waveInAddBuffer", waveInAddBuffer(m_hWaveIn,
                &m_recHdr[m_recIdx], sizeof(WAVEHDR)));
            checkResult("waveInStart", waveInStart(m_hWaveIn));
        }
        m_recIdx = (m_recIdx + 1) % REC_BUFFER_NR;
    }

    return res;
}

//
:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::
// ::
// :: Function name: AudioManager::closeAudioOut()
// ::
// :: Function Parameters: NONE
// ::
// :: Function Return: NONE
// ::
// :: Funtion Purpose: stops playback and closes the given waveform-
audio output device.

// ::
// :: Notes:
// ::
// ::
// ::
//
:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::
void AudioManager::closeAudioOut()
{
      TRACE0("Wait Started\r\n");
```

```
    m_pAudioOutThread->Wait(1000);
    TRACE0("Wait Ended\r\n");

    MYTRACE0("Closing audio out");
    if (m_hWaveOut != NULL)
  {
     waveOutReset(m_hWaveOut);
     while(reclaimPlayWaveHeaders() == true)
     {
     }
     waveOutClose(m_hWaveOut);

     m_hWaveOut = NULL;
  }
}

//
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::
// ::
// :: Function name: AudioManager::openAudioOut()
// ::
// :: Function Parameters: NONE
// ::
// :: Function Return: NONE
// ::
// :: Funtion Purpose: Opens the waveform-audio output device for
playback.

// ::
// :: Notes:
// ::
// ::
// ::
//
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::
void AudioManager::openAudioOut()
{
    closeAudioOut();

    MYTRACE0("Opening audio out");
      WAVEFORMATEX wfx;
    MMRESULT mmresult;

    wfx.cbSize = 0;
    wfx.nBlockAlign = 2;
    wfx.nAvgBytesPerSec = 8000 * wfx.nBlockAlign;
    wfx.nChannels = 1;
    wfx.nSamplesPerSec = 8000;
    wfx.wBitsPerSample = 2 * 8;
    wfx.wFormatTag = WAVE_FORMAT_PCM;

    mmresult = waveOutOpen(
                    &m_hWaveOut,
                    WAVE_MAPPER,
                    &wfx,
                    NULL,
                    NULL,
                    CALLBACK_NULL );
```

```
    checkResult("waveOutOpen", mmresult);
}

//
:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::
// ::
// :: Function name: AudioManager::playAudio()
// ::
// :: Function Parameters:
// ::
// :: Function Return: NONE
// ::
// :: Funtion Purpose:
// ::
// ::
// :: Notes:
// ::
// ::
// ::
//
:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::
void AudioManager::playAudio(AudioStream* stream)
{

      TRACE(_T("Adding stream:"));
      if (!stream->isQueued())
      {
            slock.Lock();
            stream->setQueued(true);
            playList.AddTail(stream);
            slock.Unlock();
      }

}



//
:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::
// ::
// :: Function name: AudioManager::playAudio()
// ::
// :: Function Parameters:
// ::
// :: Function Return: NONE
// ::
// :: Funtion Purpose:
// ::
// ::
// :: Notes:
// ::
// ::
// ::
//
:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::
void AudioManager::saveAudio(AudioStream* stream)
{
```

```
}



//
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::
// ::
// :: Function name: AudioManager::clearAllAudioFrames()
// ::
// :: Function Parameters:
// ::
// :: Function Return: NONE
// ::
// :: Funtion Purpose:
// ::
// ::
// :: Notes:
// ::
// ::
// ::
//
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::
void AudioManager::clearAllAudioFrames()
{
	int itCount = 0;
	while (m_iterationLock)
	{
		Sleep(10);
		//TRACE(".");
		if (itCount > 50)
		{
			//MYTRACE0("FATAL ERROR: possible deadlock detected\r
\n");
			ASSERT(true);
		}
	}

	m_iterationLock = true;

	lastRxAudioSeq = 0;
/* FIXMK
	while (m_g723audioData.GetCount() > 0)
      {
	  AudioData* frame = m_g723audioData.GetHead();
	  m_g723audioData.RemoveHead();
	  delete frame;
      }
*/
	m_iterationLock = false;
}

//
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::
// ::
// :: Function name: AudioManager::addAudioFrame()
// ::
```

```
// :: Function Parameters:
// ::
// :: Function Return: NONE
// ::
// :: Funtion Purpose:
// ::
// ::
// :: Notes:
// ::
// ::
// ::
//
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::
void AudioManager::addAudioFrame(int seq, char* frameBuf, int frameLen)
{
	ASSERT(m_pCurrentStream != NULL);

	m_pCurrentStream->addFrame(seq, frameBuf, frameLen);

	if (!m_pCurrentStream->isStreaming())
	{
		if(m_pCurrentStream->isQueued())
		{
			m_pCurrentStream->setStreaming(true);
		}
	}
}

//
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::
// ::
// :: Function name: AudioManager::decodeAndPlay()
// ::
// :: Function Parameters:
// ::		pEncData - pointer to Audio buffer
// ::		len - lenght of buffer
// ::		codec (default) ENCODING_G723 - Type of audio encoding
pEncData
// ::					points to.
// ::
// :: Function Return: NONE
// ::
// :: Funtion Purpose: Decode encoded data and play
// ::
// :: Notes:
// ::
// ::
// ::
//
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::
void AudioManager::decodeAndPlay(char* pEncData, int len, int codec =
ENCODING_G723)
{
    char* pDecodedData = NULL;
    WAVEHDR* pWavehdr;
    int i, decodOffset, encodeOffset;

#if defined(_WIN32_WCE)
```

```
    if (m_hWaveIn == NULL && m_hWaveOut == NULL)
    {
        openAudioOut();
    }

    if (m_hWaveOut == NULL)
    {
        TRACE(_T("Received message while recording throwing out\r\n"));
        return;
    }
#endif

    if (codec == ENCODING_G723)
    {
        decodOffset = encodeOffset = 0;
        pDecodedData = new char[BUFFER_LENGTH];
        VERIFY(len == G723_BUFFER_LENGTH);

        int lineSize = 0;
            for (i = 0; i < G723_FRAMES; i++)
        {
            lineSize = GetLineSize(pEncData + encodeOffset);

                  if (lineSize == 24)
                  {
                        MYTRACE0("ERROR line size was 24. Expected 20");
                  }
                  else
                  {
                        VERIFY(lineSize == 20);
                  }


#if defined(_WIN32_WCE)
            Decod((short*)(pDecodedData + decodOffset),
                pEncData + encodeOffset, 0);
#else
            Decod(m_pG723DataBuf, pEncData + encodeOffset, 0);
            Write_lbc_buf(m_pG723DataBuf, Frame,
                (short*)(pDecodedData + decodOffset));
#endif

            decodOffset += 480;
            encodeOffset += 20;
        }

        len = BUFFER_LENGTH;
    }
    else
    {
        VERIFY(!"Unknown codec");
    }

    if (pDecodedData != NULL)
    {
        ASSERT(m_hWaveOut!=NULL);
            pWavehdr = new WAVEHDR;
        memset(pWavehdr, 0, sizeof(WAVEHDR));

        pWavehdr->lpData = pDecodedData;
        pWavehdr->dwBufferLength = len;
```

```
        checkResult("waveOutPrepareHeader",
            waveOutPrepareHeader(m_hWaveOut, pWavehdr, sizeof
(WAVEHDR)));

        checkResult("waveOutWrite",
            waveOutWrite(m_hWaveOut, pWavehdr, sizeof(WAVEHDR)));

        m_playWaveHeaders.AddTail(pWavehdr);
    }
}

//
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::
// ::
// :: Function name: AudioManager::checkResult()
// ::
// :: Function Parameters:
// ::       name - String indicating MM
// ::       result - MMRESULT to check
// ::
// :: Function Return: NONE
// ::
// :: Funtion Purpose: Thows assert if MMSYSERR_NOERROR was not received
// ::
// :: Notes:
// ::          (0)  MMSYSERR_NOERROR      no error
// ::          (1)  MMSYSERR_ERROR        unspecified error
// ::          (2)  MMSYSERR_BADDEVICEID  device ID out of range
// ::          (3)  MMSYSERR_NOTENABLED   driver failed enable
// ::          (4)  MMSYSERR_ALLOCATED    device already allocated
// ::          (5)  MMSYSERR_INVALHANDLE  device handle is invalid
// ::          (6)  MMSYSERR_NODRIVER     no device driver present
// ::          (7)  MMSYSERR_NOMEM        memory allocation error
// ::          (8)  MMSYSERR_NOTSUPPORTED function isn't supported
// ::          (9)  MMSYSERR_BADERRNUM    error value out of range
// ::          (10) MMSYSERR_INVALFLAG    invalid flag passed
// ::          (11) MMSYSERR_INVALPARAM   invalid parameter passed
// ::          (12) MMSYSERR_HANDLEBUSY   handle being used
// ::          (13) MMSYSERR_INVALIDALIAS specified alias not found
// ::          (14) MMSYSERR_BADDB        bad registry database
// ::          (15) MMSYSERR_KEYNOTFOUND  registry key not found
// ::          (16) MMSYSERR_READERROR    registry read error
// ::          (17) MMSYSERR_WRITEERROR   registry write error
// ::          (18) MMSYSERR_DELETEERROR  registry delete error
// ::          (19) MMSYSERR_VALNOTFOUND  registry value not found
// ::          (20) MMSYSERR_NODRIVERCB   driver does not call
DriverCallback
// ::          (20) MMSYSERR_LASTERROR    last error in range
// ::
// ::          (33) WAVERR_STILLPLAYING   There are still buffers in the
queue
// ::
//
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::
void AudioManager::checkResult(const char* name, MMRESULT result)
{

    if (VERIFY_VALS)
      {
```

```
            //CString* threadId = new CString("TEST ");
            //threadId->Format(_T("MM %d"), result);
            //pMessageManager->FireOnAudioEcho(0, threadId, NULL);

            VERIFY(result == MMSYSERR_NOERROR && name != NULL);
      }
}


//
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::
// ::
// :: Function name: AudioManager::playTestFrame()
// ::
// :: Function Parameters: NONE
// ::
// :: Function Return: NONE
// ::
// :: Funtion Purpose: Test that Audio Out is working properly
// ::
// :: Notes:
// ::
// ::
// ::
//
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::
void AudioManager::playTestFrame()
{
      int i=0;

      // Play test
      char* pBuffer = NULL;

    static BYTE buffer1[] = MTALK_TEST_FRAME;

    pBuffer = (char*)&buffer1;

    if (m_hWaveOut != NULL && true)
    {
        waveOutReset(m_hWaveOut);
    }

    decodeAndPlay(pBuffer, G723_BUFFER_LENGTH);

    if (true)
    {
        Sleep(G723_FRAME_TIME);
    }

}

//
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::
// ::
// :: Function name: AudioManager::getBufferCount()
// ::
// :: Function Parameters: NONE
```

```
// ::
// :: Function Return: NONE
// ::
// :: Funtion Purpose: Indicate number of Record Buffers waiting to send
// ::
// :: Notes:
// ::
// ::
// ::
//
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::
int AudioManager::getBufferCount()
{
	ASSERT(m_pClientsList != NULL);
	return m_pClientsList->m_sendBufferList.GetCount();
}

//
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::
// ::
// :: Function name: AudioManager::getNextBuffer()
// ::
// :: Function Parameters: NONE
// ::
// :: Function Return:
// ::		BufferNode for next audio frame
// ::
// :: Funtion Purpose: Indicate number of Record Buffers waiting to send
// ::
// :: Notes:
// ::
// ::
// ::
//
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::
BufferNode*  AudioManager::getNextBuffer()
{
	ASSERT(m_pClientsList != NULL);
	return m_pClientsList->m_sendBufferList.GetHead();
}

//
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::
// ::
// :: Function name: AudioManager::releaseBuffer()
// ::
// :: Function Parameters: NONE
// ::
// :: Function Return:
// ::		BufferNode for next audio frame
// ::
// :: Funtion Purpose: Indicate number of Record Buffers waiting to send
// ::
// :: Notes:
// ::
// ::
// ::
```

```
//
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::
void AudioManager::releaseBuffer()
{
      ASSERT(m_pClientsList != NULL);
      m_pClientsList->m_sendBufferList.RemoveHead();
}

//
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::
// ::
// :: Function name: transferAudioOut
// ::
// :: Function Parameters:
// ::       pAudioManager - A pointer to AudioManager
// ::
// :: Function Return: NONE
// ::
// :: Funtion Purpose: Send Audio to Server
// ::
// :: Notes:
// ::
// ::
// ::
//
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::
static unsigned int transferAudioOut(void* pAudioManager)
{
      ASSERT(pAudioManager != NULL);
      AudioManager* pAudMgr = (AudioManager*)pAudioManager;

      BufferNode* pBN;
      bool sending = false;

      while(pAudMgr->isRecording() || (pAudMgr->getBufferCount() > 0)
              && !pAudMgr->isShuttingDown())
      {

            if (pAudMgr->getBufferCount() > 0)
        {


            pBN = pAudMgr->getNextBuffer();

                  // Send the Audio Frame
                  AudioMessage msg(pBN->m_pSharedNode->m_pData,
                        pBN->m_pSharedNode->m_len - pBN->m_offset);

                  msg.setAudioSeqLen(pAudMgr->m_txAudioSeq++);
                  sending = true;

                  pMessageManager->sendMessage(&msg);


                  pAudMgr->releaseBuffer();
                  delete pBN;
        }
            else if (pAudMgr->isRecording())
```

```
                {
                        Sleep(100);
                        pAudMgr->checkRecordWaveHeaders();

                }

        }

        if (pAudMgr->getBufferCount() > 0)
      {
          pBN = pAudMgr->getNextBuffer();
              delete pBN;
      }

        if (sending)
        {
                //send EndAudio
                EndAudioMessage endAudio;
                pMessageManager->sendMessage(&endAudio);

        }

        pAudMgr->closeWaveIn();
      g_outThreadRunning = false;

        return 0;
}

//
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::
// ::
// :: Function name: playAudioStreams
// ::
// :: Function Parameters:
// ::
// :: Function Return: NONE
// ::
// :: Funtion Purpose:
// ::
// :: Notes:
// ::
// ::
// ::
//
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::
static unsigned int playAudioStreams(void* pAudioManager)
{
        ASSERT(pAudioManager != NULL);
        AudioManager* pAudMgr = (AudioManager*)pAudioManager;

        AudioStream* stream;
        while( !pAudMgr->isShuttingDown() )
        {
                if (playList.GetCount() > 0)
                {
                        slock.Lock();
                        stream = playList.GetHead();
                        playList.RemoveHead();
                        slock.Unlock();
```

```
			if (stream == NULL)
				continue;

			CString streamInfo;
			streamInfo.Format(_T("---- Playing id: %d %d\r\n"),
stream->getId(), stream->isCompleteAudio());
			MYTRACE0(streamInfo);

			if (stream->isCompleteAudio())
			{
				MYTRACE0(_T("Start playing full stream"));
				slock.Lock();
				playNonStream(pAudioManager, stream);
				slock.Unlock();
				MYTRACE0(_T("Finish playing full stream"));
			}
			else
			{
				MYTRACE0(_T("Start playing stream"));
				slock.Lock();
				playStream(pAudioManager, stream);
				slock.Unlock();
				MYTRACE0(_T("Finish playing stream"));
			}
			stream->setQueued(false);
			stream = NULL;


		}
		else
		{
			DoEvents1();
			Sleep(100);
		}
	}

	return 0;
}

static void playNonStream(void* pAudioManager, AudioStream* stream)
{
	ASSERT(stream->getFrames() != NULL);
	AudioManager* pAudMgr = (AudioManager*)pAudioManager;

	MYTRACE0("Start playing full stream");

	CList<AudioData*, AudioData*>* g723audioData = stream->getFrames
();

	POSITION pos = g723audioData->GetHeadPosition();
	int count =  g723audioData->GetCount();

	for (int i=0; i < count; i++)
	{
		AudioData* frame = g723audioData->GetNext(pos);
		pAudMgr->decodeAndPlay(frame->m_audioFrame, frame->
m_frameLen, ENCODING_G723);
	}
	MYTRACE0("Finished play full stream");
}
```

```
/**
  * Todo determine max jitter and latency allowed
 **/
static void playStream(void* pAudioManager, AudioStream* stream)
{
      AudioManager* pAudMgr = (AudioManager*)pAudioManager;
      MYTRACE0("Start playing stream");

      int latencyCoef = 20;
      CList<AudioData*, AudioData*>* g723audioData = stream->getFrames
();

      int timeout = 0;

      // try and buffer up at least two frame for our jitter buffer
      while (g723audioData->GetCount() <= 1 && !stream->isCompleteAudio
())
      {
            if (timeout++ > latencyCoef)
            {
                  MYTRACE0("Starved while buffering");
                  return;
            }

            DoEvents1();
            Sleep(G723_FRAME_TIME/4);
      }

      bool starvation = false;
      bool hasLooped= false;
      int timout = 0;
      int prevCount = 0;
      int count = g723audioData->GetCount();

      POSITION pos = g723audioData->GetHeadPosition();
      while(count > 0)
      {
            count = g723audioData->GetCount();

            if (timeout++ > latencyCoef)
            {
                  MYTRACE("Starved while streaming: %d", timeout);
                  break;
            }

            if (prevCount < count)
            {
                  if (hasLooped)
                  {
                        g723audioData->GetNext(pos);
                        hasLooped = true;
                  }

                  // play any frames we have not already played
                  for (int i = 0; i < (count - prevCount) && pos !=
NULL; ++i)
                  {
                        AudioData* frame = g723audioData->GetNext(pos);
                        MYTRACE("Queueing frame: %d", frame->
m_sequence);
```

```
                        pAudMgr->decodeAndPlay(frame->m_audioFrame,
frame->m_frameLen, ENCODING_G723);
                  }


                  if (pos == NULL)
                  {
                        pos = g723audioData->GetTailPosition();
                        hasLooped = true;
                  }

                  prevCount = count;
                  timeout = 0;
                  starvation = false;
            }
            else
            {
                  starvation = true;
            }

            if (stream->isCompleteAudio())
            {
                  MYTRACE0("Play stream completed");
                  break;
            }

            if (!starvation)
            {
                  DoEvents1();
                  Sleep(G723_FRAME_TIME * (count - prevCount));
            }
            else
            {
                  DoEvents1();
                  Sleep(G723_FRAME_TIME/4);
            }
      }

      MYTRACE0("Finished play stream");
}

void DoEvents1()
{
    MSG msg;

    // Process existing messages in the application's message queue.
    // When the queue is empty, do clean up and return.
    while (::PeekMessage(&msg,NULL,0,0,PM_NOREMOVE))
    {
         if (!AfxGetThread()->PumpMessage())
              return;
    }
}

//
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::
// ::
// :: Function name: newAudioStream
// ::
// :: Function Parameters:
```

```
// ::
// :: Function Return: a new audio stream
// ::
// :: Funtion Purpose: create a new audio stream to store incoming voice
msg
// ::
// :: Notes:
// ::
// ::
// ::
//
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::
AudioStream* AudioManager::newAudioStream()
{
	m_pCurrentStream = m_pAudioCollection->newStream();
	return m_pCurrentStream;
}

//
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::
// ::
// :: Function name: isPlaying
// ::
// :: Function Parameters:
// ::
// :: Function Return: a playing status
// ::
// :: Funtion Purpose: return playing status
// ::
// :: Notes:
// ::
// ::
// ::
//
::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:::
bool AudioManager::isPlaying()
{
	if(m_isPlaying)
	{
	}
	else
	{
	}

	return m_isPlaying;
}

void AudioManager::setPlayStatus(bool status)
{
	m_isPlaying = status;
}

void AudioManager::setReceivedCompleteAudio()
{
	ASSERT(m_pCurrentStream != NULL);

	m_pCurrentStream->setCompleted();
}
```